Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Лабораторна робота №5

з дисципліни

«Алгоритми та структури даних»

Виконав: студент групи ІМ-42

Максим Крамаренко Юрійович

номер варіанту: 17

Перевірив:

Сергієнко А. М.

Київ 2025

Постановка задачі:

1. Представити напрямлений граф із заданими параметрами так само, як у лабораторній роботі №3. Відмінність: коефіцієнт k = 1.0 - n3*0.01 - n4*0.005 - 0.15.
Отже, матриця суміжності Adir напрямленого графа за варіантом формується таким чином:

1) встановлюється параметр (seed) генератора випадкових чисел, рівне номеру варіанту n1n2n3n4;

2) матриця розміром n nзаповнюється згенерованими випадковими числами в діапазоні [0, 2.0);

3) обчислюється коефіцієнт k = 1.0 - n3*0.01 - n4*0.005 - 0.15, кожен елемент матриці множиться на коефіцієнт k;

4) елементи матриці округлюються: 0 — якщо елемент менший за 1.0, 1 — якщо елемент більший або дорівнює 1.0.

2. Створити програму, яка виконує обхід напрямленого графа вшир (BFS) та вглиб (DFS).

- обхід починати з вершини із найменшим номером, яка має щонайменше одну вихідну дугу;
- при обході враховувати порядок нумерації;
- у програмі виконання обходу відображати покроково, черговий крок виконувати за натисканням кнопки у вікні або на клавіатурі.

3. Під час обходу графа побудувати дерево обходу. У програмі дерево обходу виводити покроково у процесі виконання обходу графа. Це можна виконати одним із двох способів:

- або виділяти іншим кольором ребра графа;
- або будувати дерево обходу поряд із графом.

4. Зміну статусів вершин у процесі обходу продемонструвати зміною кольорів вершин, графічними позначками тощо, або ж у процесі обходу виводити протокол обходу у графічне вікно або в консоль.

5. Якщо після обходу графа лишилися невідвідані вершини, продовжувати обхід з невідвіданої вершини з найменшим номером, яка має щонайменше одну вихідну дугу.

При проєктуванні програми також слід врахувати наступне:

1) мова програмування обирається студентом самостійно;

2) графічне зображення усіх графів має формуватися програмою з тими ж вимогами, як у ЛР №3;

3) всі графи обов’язково зображувати у графічному вікні;

4) типи та структури даних для внутрішнього представлення всіх даних у програмі слід вибрати самостійно.

Варіант 17:

n1 = 4
n2 = 2
n3 = 1
n4 = 7

SEED = 4217

n = n3 + 10

Текст програми:

Файл №1 (main.py)

```
from graph_utils import draw_graph
import random
import math

n1 = 4
n2 = 2
n3 = 1
n4 = 7

n = n3 + 10

k = 1.0 - n3*0.01 - n4*0.005 - 0.15

random.seed(4217)

def print_matrix(matrix, title="Matrix", line_length=1):
    print(f"\n=== {title} ===")
    for row in matrix:
        print(" ".join(f"{num:{line_length}}" for num in row))  # Each number is
line_length characters wide
    print()

def get_dir(n, k):
    result = [[0] * n for _ in range(n)]
```

```
    for i in range(n):
        for j in range(n):
            result[i][j] = math.floor(random.uniform(0, 2.0) * k)
    return result

def bfs_tree(graph, start):
    visited = [False] * len(graph)
    levels = [-1] * len(graph)  # Track BFS level for each node
    queue = []
    queue.append(start)
    visited[start] = True
    levels[start] = 0  # Starting node is at level 0
    bfs_tree = [[0] * len(graph) for _ in range(len(graph))]
    steps = []  # Collect steps for visualization

    current_level = 0
    nodes_at_current_level = 1  # Count of nodes at the current level
    nodes_at_next_level = 0     # Count of nodes at the next level

    while queue:
        steps.append({
            "visited": [i + 1 for i, v in enumerate(visited) if v],
            "queue": [q + 1 for q in queue],
            "levels": [levels[i] for i in range(len(levels))],
            "current_level": current_level
        })

        node = queue.pop(0)
        nodes_at_current_level -= 1

        for neighbor in range(len(graph)):
            if graph[node][neighbor] and not visited[neighbor]:
                visited[neighbor] = True
                queue.append(neighbor)
                levels[neighbor] = current_level + 1
                nodes_at_next_level += 1
                bfs_tree[node][neighbor] = 1

        # If we've processed all nodes at the current level, move to the next
level
        if nodes_at_current_level == 0:
```

```
            current_level += 1
            nodes_at_current_level = nodes_at_next_level
            nodes_at_next_level = 0

    return bfs_tree, steps

def dfs_tree(graph, start):
    visited = [False] * len(graph)
    stack = [(start, -1)]  # Store (node, parent) pairs
    dfs_tree = [[0] * len(graph) for _ in range(len(graph))]
    steps = []  # Collect steps for visualization

    while stack:
        node, parent = stack.pop()
        if not visited[node]:
            visited[node] = True

            steps.append({"visited": [i + 1 for i, v in enumerate(visited) if
v], "queue": [node + 1]})

            # Add edge from parent to this node (if parent exists)
            if parent != -1:
                dfs_tree[parent][node] = 1

            for neighbor in range(len(graph) - 1, -1, -1):  # Reverse order to
maintain DFS behavior
                if graph[node][neighbor] and not visited[neighbor]:
                    # Push neighbor and its parent (current node) to stack
                    stack.append((neighbor, node))

    return dfs_tree, steps

dir = get_dir(n, k)

bfs, bfs_steps = bfs_tree(dir, 0)
dfs, dfs_steps = dfs_tree(dir, 0)

print_matrix(dir, "Directed Graph")
print_matrix(bfs, "BFS Tree")
print_matrix(dfs, "DFS Tree")
```

```
draw_graph(dir, directed=True, title="Original Directed Graph")

print("\nBFS Vertex Numbering:")
print("Original Vertex -> New Numbering")
draw_graph(bfs, directed=True, title="BFS Tree", graph_type="bfs",
steps=bfs_steps)
print()

print("\nDFS Vertex Numbering:")
print("Original Vertex -> New Numbering")
draw_graph(dfs, directed=True, title="DFS Tree", graph_type="dfs",
steps=dfs_steps)
print()
```

Файл №2 (graph_utils.py)

```
import math
import random
import matplotlib.pyplot as plt

random.seed(4217)

def draw_graph(matrix, directed=False, title=None, graph_type=None,
steps=None):
    order = 0
    R = 10  # Radius of the circular layout
    angle_step = 2 * math.pi / (len(matrix) - 1)  # Angle between nodes
    # Calculate node positions
    positions = []
    for i in range(len(matrix)):
        if i == 0:
            positions.append((0, 0))  # Center node
            continue
        angle = i * angle_step
        x = R * math.cos(angle)
        y = R * math.sin(angle)
        positions.append((x, y))

    node_R = 0.8  # Node radius
```

```
    # Helper function to adjust for node boundary
    def adjust_for_R(x1, y1, x2, y2, offset):
        dx, dy = x2 - x1, y2 - y1
        length = math.sqrt(dx ** 2 + dy ** 2)
        if length == 0:
            return x1, y1, x2, y2
        scale = (length - offset) / length
        return x1 + dx * (1 - scale), y1 + dy * (1 - scale), x2 - dx * (1 -
scale), y2 - dy * (1 - scale)

    # Function to draw a single step
    def draw_step(step, step_index, prev_active_node=None, order=0):
        if len(matrix) > 1:
            visited = step.get("visited", [])
            queue_or_stack = step.get("queue", [])
            active_node = queue_or_stack[0] if queue_or_stack else None
            order += 1
            print(f"{active_node} -> {order}")

            # BFS specific information
            levels = step.get("levels", [-1] * len(matrix))

            plt.figure(figsize=(8, 8))
            plt.title(f"{title} - Step {step_index + 1}", fontsize=14)

            # Plot nodes
            for i, (x, y) in enumerate(positions):
                # BFS: Color by level with fixed colors per level
                if graph_type == "bfs":
                    if i + 1 == prev_active_node:
                        color = "#FF0000"  # Red for previously active node
                    elif levels[i] >= 0:
                        # Fixed colors based on level (won't change as traversal
progresses)
                        level_colors = [
                            "#FF0000",   # Level 3 - Red
                            "#FF8C00",  # Level 1 - Dark Orange
                            "#FFD700",  # Level 0 - Gold
                        ]
                        color_index = min(levels[i], len(level_colors) - 1)
                        color = level_colors[color_index]
```

```
                else:
                    color = 'lightgray'  # Unvisited
            # DFS: Use existing coloring
            else:
                if i + 1 == active_node:
                    color = "#FF6347"  # Tomato for DFS active node
                elif i + 1 in visited:
                    color = "#FFD700"  # Gold for visited nodes
                else:
                    color = 'lightgray'  # Unvisited nodes

            plt.scatter(x, y, s=500, color=color, edgecolor="black",
linewidth=1, zorder=2)
            plt.text(x, y, str(i + 1), fontsize=12, ha="center",
va="center", zorder=4)

        # Plot edges with highlighting for active paths
        for i in range(len(matrix)):
            for j in range(len(matrix)):
                if matrix[i][j]:
                    x1, y1 = positions[i]
                    x2, y2 = positions[j]
                    x1, y1, x2, y2 = adjust_for_R(x1, y1, x2, y2, node_R)

                    # For BFS, use fixed level-based edge coloring
                    if graph_type == "bfs" and levels[i] >= 0 and levels[j]
>= 0:
                        # Edges between any two levels
                        level_diff = abs(levels[i] - levels[j])
                        if level_diff == 1:  # Connections between adjacent
levels
                            edge_color = "#FF6347"  # Tomato for level
transitions
                            edge_width = 2.5
                        elif level_diff == 0:  # Connections within the same
level
                            edge_color = "#FFD700"  # Gold for same level
                            edge_width = 2.0
                        else:
                            edge_color = "gray"  # Gray for other
connections
```

```
                            edge_width = 1.0
                        else:
                            # Use the existing edge coloring logic for DFS
                            is_active_edge = (i+1 == active_node and j+1 in
visited) or (j+1 == active_node and i+1 in visited)
                            is_traversed_edge = (i+1 in visited and j+1 in
visited)

                            if is_active_edge:
                                edge_color = "#FF6347"  # Tomato for active
edges
                                edge_width = 2.5
                            elif is_traversed_edge:
                                edge_color = "#FFD700"  # Gold for traversed
edges
                                edge_width = 2.0
                            else:
                                edge_color = "gray"  # Gray for untraversed
edges
                                edge_width = 1.0

                        plt.plot([x1, x2], [y1, y2], color=edge_color,
linewidth=edge_width, zorder=1)

                        if directed:
                            plt.arrow(
                                x1, y1, x2 - x1, y2 - y1,
                                head_width=0.30, length_includes_head=True,
                                color=edge_color, linewidth=edge_width, zorder=3
                            )

            plt.xlim(-15, 15)
            plt.ylim(-15, 15)
            plt.axis("off")
            plt.show()

    # If steps are provided, draw each step
    if steps:
        prev_active_node = None
        for step_index, step in enumerate(steps):
            draw_step(step, step_index, prev_active_node, order)
```

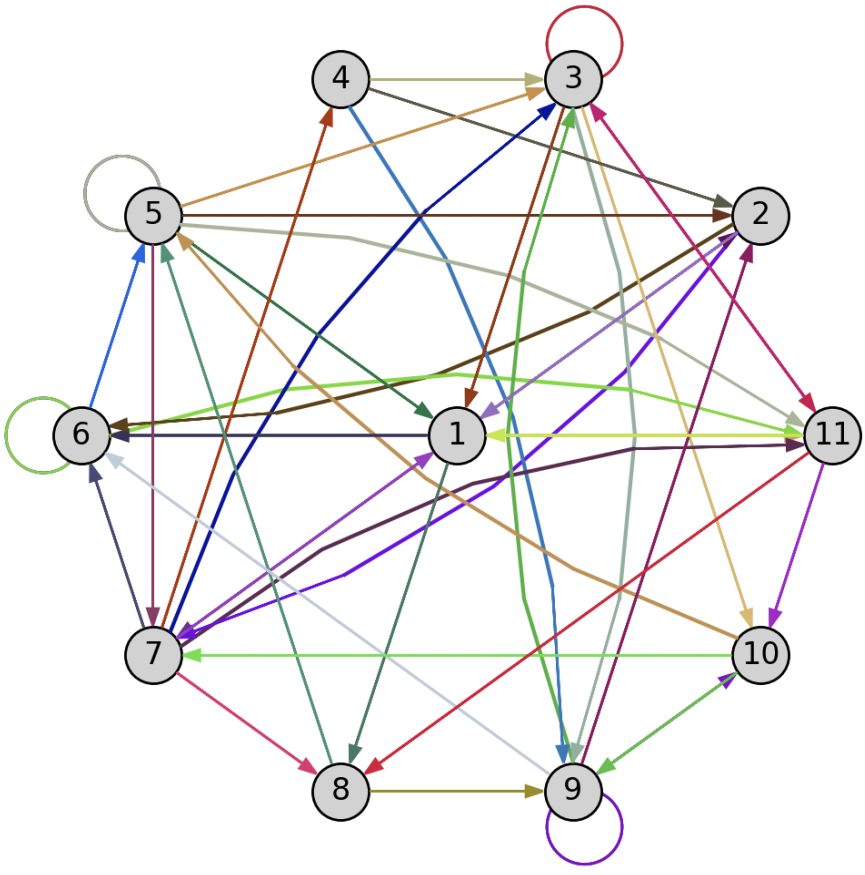
```
            order += 1
            if graph_type == "bfs":
                prev_active_node = step.get("queue", [None])[0] if
step.get("queue") else None
    else:
        plt.figure(figsize=(8, 8))
        for i, (x, y) in enumerate(positions):
          # Draw node
            plt.scatter(x, y, s=500, color='lightgray', edgecolor='black',
linewidth=1, zorder=2)
          # Label node
            plt.text(x, y, str(i + 1), fontsize=12, ha="center", va="center",
zorder=4)


        # Draw edges
        for i in range(len(matrix)):
            for j in range(len(matrix)):
                if matrix[i][j]:
                    # Choose edge color based on graph type or random
                    edge_color = (random.randint(0, 235) / 255,
random.randint(0, 235) / 255, random.randint(0, 235) / 255)

                    # Draw self-loop if a node connects to itself
                    if matrix[i][i]:
                        x, y = positions[i]
                        loop_radius = 1
                        if(x != 0 and y != 0):
                            vector_length = math.sqrt(x ** 2 + y ** 2)
                            x += x * loop_radius / vector_length
                            y += y * loop_radius / vector_length
                        else:
                            y += loop_radius
                        loop = plt.Circle((x, y), loop_radius, color=edge_color,
fill=False, zorder=1)
                        plt.gca().add_patch(loop)

                    x1, y1 = positions[i]
                    x2, y2 = positions[j]

                    dx, dy = x2 - x1, y2 - y1
```

```
                    length = math.sqrt(dx ** 2 + dy ** 2)
                    x1, y1, x2, y2 = adjust_for_R(x1, y1, x2, y2, node_R)

                    # Draw edges that would pass through center
                    if (length >= 2*(R - node_R) and length <= 2*(R + node_R)):
                        norm_dx, norm_dy = dx / length, dy / length
                        perp_x, perp_y = -norm_dy, norm_dx
                        curve_factor = 3.0 + random.uniform(-0.5, 0.5)
                        midx = (x1 + x2) / 2 + perp_x * curve_factor
                        midy = (y1 + y2) / 2 + perp_y * curve_factor
                        t_values = [0, 0.25, 0.5, 0.75, 1.0]
                        curve_x = []
                        curve_y = []

                        for t in t_values:
                            bx = (1-t)**2 * x1 + 2*(1-t)*t * midx + t**2 * x2
                            by = (1-t)**2 * y1 + 2*(1-t)*t * midy + t**2 * y2
                            curve_x.append(bx)
                            curve_y.append(by)

                        if directed:
                            plt.plot(curve_x[:-1], curve_y[:-1],
color=edge_color, zorder=3)
                            last_segment_x = curve_x[-2]
                            last_segment_y = curve_y[-2]
                            plt.arrow(last_segment_x, last_segment_y,
                                    curve_x[-1] - last_segment_x,
                                    curve_y[-1] - last_segment_y,
                                    head_width=0.30, length_includes_head=True,
                                    color=edge_color, zorder=4)
                        else:
                            plt.plot(curve_x, curve_y, color=edge_color,
zorder=3)
                        continue

                    # Draw direct edges
                    if directed:
                        plt.arrow(x1, y1, x2 - x1, y2 - y1, head_width=0.30,
length_includes_head=True, color=edge_color, zorder=4)
                        continue
                    plt.plot([x1, x2], [y1, y2], color=edge_color, zorder=3)
```

```
        plt.xlim(-15, 15)
        plt.ylim(-15, 15)
        plt.axis("off")
        plt.show()
```

Тестування програми:

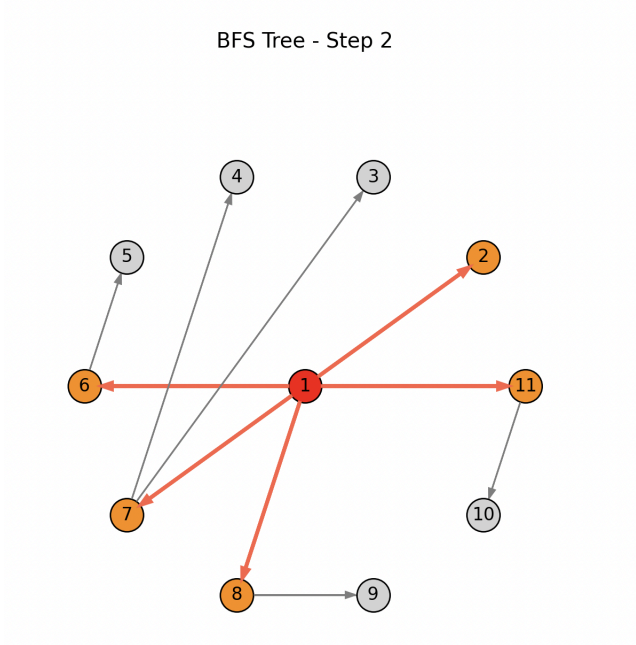
```
=== Directed Graph ===
0 1 0 0 0 1 1 1 0 0 1
1 0 0 0 0 1 1 0 0 0 0
1 0 1 0 0 0 0 0 1 1 1
0 1 1 0 0 0 0 0 1 0 0
1 1 1 0 1 0 1 0 0 0 1
0 0 0 0 1 1 0 0 0 0 1
1 0 1 1 0 1 0 1 0 0 1
0 0 0 0 1 0 0 0 1 0 0
0 1 1 0 0 1 0 0 1 1 0
0 0 0 0 1 0 1 0 1 0 0
1 0 1 0 0 0 0 1 0 1 0
```

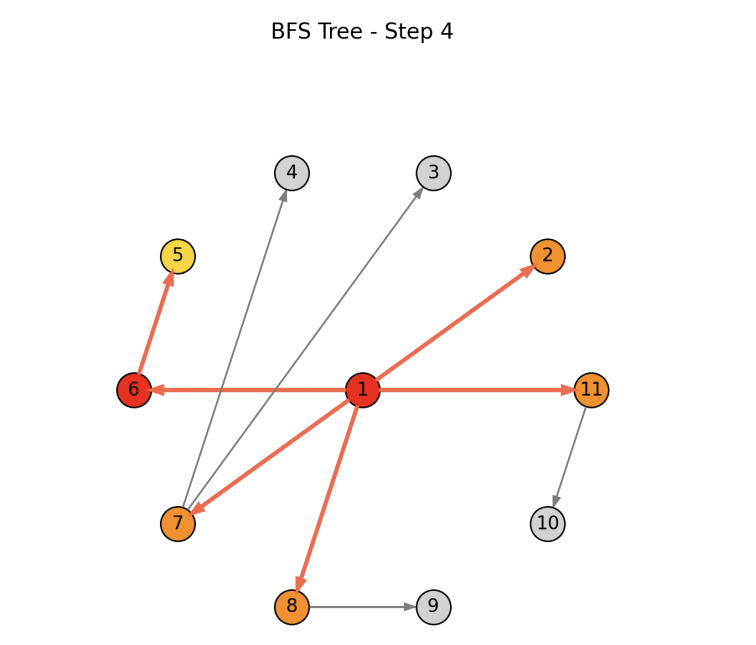


Обхід напрямленого графа вшир (BFS)

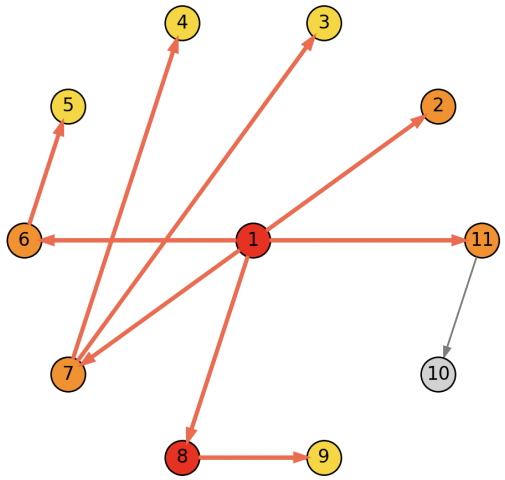
```
=== BFS Tree ===
0 1 0 0 0 1 1 1 0 0 1
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0
```

```
BFS Vertex order:
Vertex number -> Order number
1 -> 1
2 -> 2
6 -> 3
7 -> 4
8 -> 5
11 -> 6
5 -> 7
3 -> 8
4 -> 9
9 -> 10
10 -> 11
```

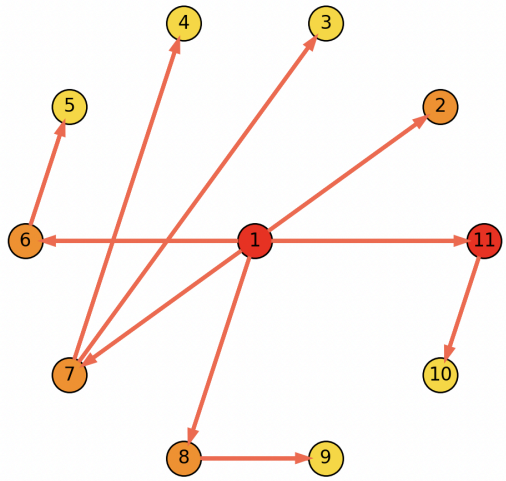




BFS Tree - Step 6



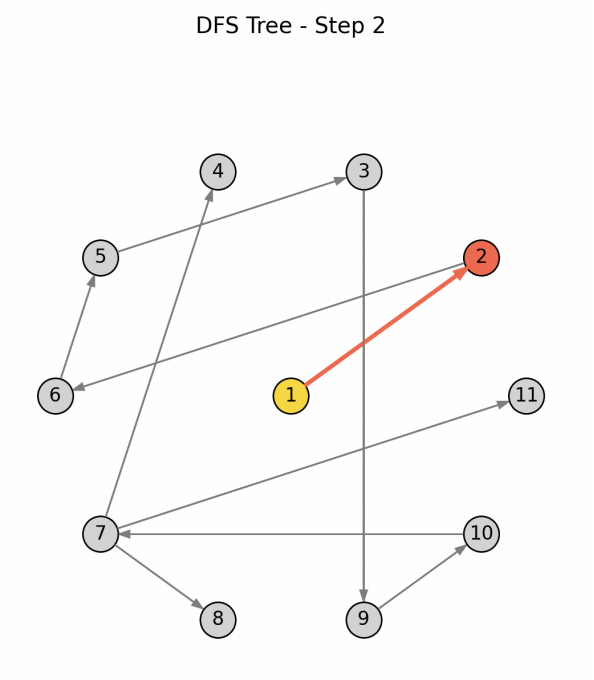
BFS Tree - Step 7

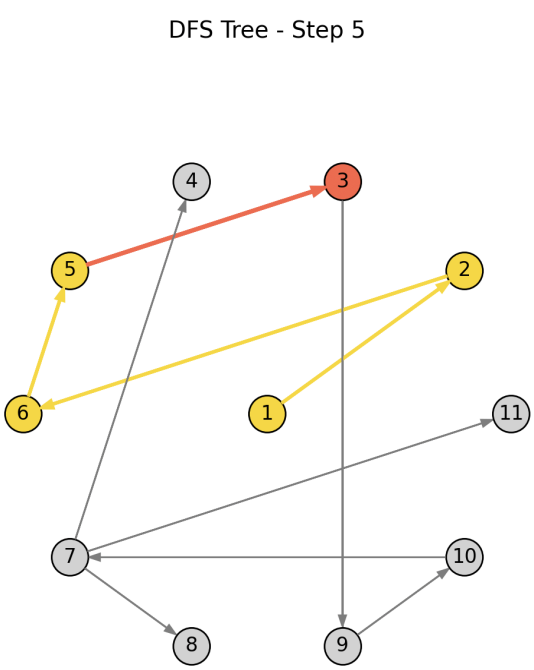


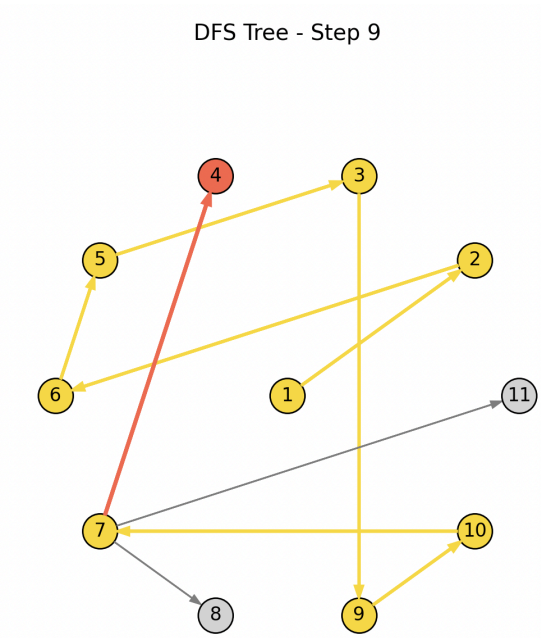
Обхід напрямленого графа вглиб (DFS)

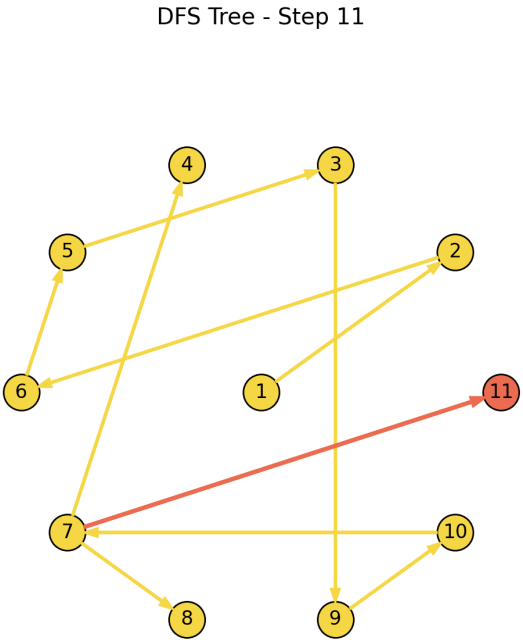
```
=== DFS Tree ===
0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
```

```
DFS Vertex order:
Vertex number -> Order number
1 -> 1
2 -> 2
6 -> 3
5 -> 4
3 -> 5
9 -> 6
10 -> 7
7 -> 8
4 -> 9
8 -> 10
11 -> 11
```









Висновок:

Засвоїв теоретичний матеріал лекцій та отримав практичний досвід у створенні функцій обходу графів в ширину і глиб, та набув навичок у їх аналізуванні. Покращив навички роботи з графічними вікнами та функціями для них. Збільшив досвід роботи з матрицями та створення функцій для їх аналізу.